Implementation and Testing: Cache Oblivious/Aware Algorithms

Richard Wu, York Liu, Alex Huang

21 Dec 2020

0 Introduction

In recent days, many algorithms and applications tend to operate on big data that is too large to be stored entirely in cache or even in the main memory. Operations on the data which is located farther in the memory hierarchy are noticeably slower and more expensive. For example, an operation in CPU register files is usually one thousand times faster than loading an item from main memory, and one million times faster if the data is located in the secondary disk. To investigate on the I/O cost of an algorithm, we usually use the ideal cache model to derive an upper bound of an algorithm's I/O complexity. Also, many algorithms have been proposed, in addition to the traditional time and space complexity, to have a reasonable and decent I/O performance. Some algorithms, classified as cache aware algorithms, may need the parameters of the cache model as inputs, such as the block (line) size and the capacity, whereas some others do not, i.e. given the ideal cache model, the theoretical bound applies on any cache parameter, and these algorithms are classified as cache oblivious.

In this project, to investigate on the cache aware/oblivious algorithms, we implemented a cache simulator with adjustable cache replacement policy, (LRU, RR, FIFO, MRU, NMRU), adjustable associativity, block size, and number of sets. Based on this cache simulator, we also implemented several cache oblivious and cache aware algorithms discussed in class regarding sorting and matrix multiplication. For sorting, we implemented Merge Sort, K-way Merge Sort, and Funnel Sort. For matrix multiplication, we implemented Blocked Multiplication and Z-layout Multiplication. We tested and analyzed their performance on the ideal cache model, i.e fully associative, LRU (to approximate the optimal Belady's replacement policy), and also on the practical L1 cache parameters used in real-world CPUs, including Pentium III, MIPS 10000, AMD Athlon, and Itanium 2. We tested the algorithms on different problem size and cache parameters and compared their performance. This report will discuss the implementation details, experimental results, and our analysis.

1 Cache Simulator

Given an algorithm, there are four steps to evaluate the algorithm's I/O performance under our cache simulation. First, we wrap up the function of the algorithm under a main function to create an executable. Second, we run Valgrind Lackey, a Linux tool, on the executable to get its memory access history. Next, we filter the history and only keep the relevant operations. Finally, we feed the history of the relevant operations to the cache simulator to calculate the number of hits, the number of misses, the number of evictions, and the miss rate, under different cache parameters and replacement policies. The steps are illustrated in the figure below (an example of Selection Sort):

Figure 1: Four steps to evaluate an algorithm's I/O performance

1.1 Wrap up the function

We are given a function of an algorithm, for example, Selection Sort in the figure above. Now, in the main function, we do the initialization: create an array using malloc, initialize the array randomly, make a copy of it for the validation at the end, etc. Then we write two markers, MARKER_START and MARKER_END, before calling the function and after calling the function. The markers are global variables and their addresses will be stored (in a file) for filtration in step 3. During the filtration, we will filter out all the irrelevant instructions: all the instructions before MARKER_START and all the instructions after MARKER_END. After the MARKER_END, we will validate the correctness of the algorithm, i.e. the numbers are successfully sorted, and we will do the cleanup. In the function of the algorithm itself, all the data structures used, for example, temporary arrays for Merge Sort and K-way Merge Sort, recursive funnels for Funnel Sort, must be dynamically allocated using malloc. This will help the filtration, because we need to filter out all the stack operations. Details will be provided in the subsection 1.3. Then, we compile the program with flag -O0, meaning that there is no compiler optimization.

1.2 Run Valgrind Lackey

Second we run Valgrind Lackey on the executable compiled by issuing the command:

```
$ valgrind --tool=lackey --trace-mem=yes --log-fd=1 -v ./selectionSort -N 1000 > trace.tmp
```

Valgrind will produce the history of memory access of this program. The file trace.tmp will contain thousands of lines where each line is in the format:

$$(I, \sqcup S, \sqcup L) \sqcup address, size$$

where $\sqcup$ stands for space; I stands for instruction load; S stands for data store; L stands for data load; *address* contains the memory address of the access; *size* represents the size (width) of the memory access, usually 4 or 8 for S or L and 1 to 8 for I because we are using CISC. Next, we will feed the output of Valgrind Lackey to a filter program.

1.3 Filtration

During the filtration step, we will filter out all the irrelevant memory access. The history of memory access before the filtration and after the filtration are illustrated in the figure below:

Figure 2: The history of memory access before and after the filtration

First, we filter out all the instruction loads, i.e. all the lines starting with I. This is reasonable to do because in modern machines, instruction cache (icache) and data cache (dcache) are usually separated in L1 level, otherwise there will be structural hazards in the pipeline. In our simulation, we will only consider the cache performance of the algorithm on the data structure it manipulates.

Second, we filter out all data loads and stores from stack, i.e. all the lines starting with $\sqcup S$ or $\sqcup D$ with *address* starting with 0x1ffe. The stack access is from two sources: 1. push and pop instructions for a function's stack frame. 2. temporary variables that are not entirely manipulated in registers, such as the int i in the line for(int $i = 0$; $i <$ N - 1, i++); or *mid*, *start*, *end* used in Merge Sort and Funnel Sort. This is reasonable to do because we are evaluating the logic of the algorithm itself, instead of the technical details involved in compiling and optimization. This also explains why all the temporary data structures used in the algorithm must be dynamically allocated instead of statically allocated in the stack to avoid being filtered.

Finally, as mentioned, we need to filter out all the instructions before MARKER_START and all the instructions after MARKER_END. This is easy to do because we have already recorded the address of the markers,

i.e. ␣S␣ &MARKER_START, 4 corresponds to MARKER_START = 123456

1.4 Feed to the cache simulator

Given the filtered memory access history, the cache simulator we write will simulate the loads and stores in the cache. We run the simulator with the cache parameters as arguments: associativity, block (line) size, the number of sets, and replacement policy. It will evict a line if the corresponding set is full based on the cache replacement policy we set. The options include:

LRU: Least Recently Used
RR: Random Replacement
FIFO: First In First Out
MRU: Most Recently Used
NMRU: Not Most Recently Used (evict a random line other than the most recently used line)

An example of running the filtered history of Selection Sort looks like the following:

```
=======================================(set=32, E=2, L=32, r=LRU)=======================================
Function 0:[Selection Sort]
num of hits:    964790
num of misses:  38207
miss rate:      0.038093
evictions:      38146
```

Figure 3: Selection Sort run on the simulator

Where the number of sets is set to 32, associativity is set to 2, block (line) size is set to 32 bytes, and replacement policy LRU. The array being sorted has size 1000, randomly initialized.

Also the cache model we build in the simulation only has one layer, i.e. L1. Consider the average time of a memory access, $t_{average} = t_{L1hit} + \%_{L1miss} \times t_{L1miss}$, where t_{L1hit} is the time to handle a hit in L1, $\%_{L1miss}$ is the miss rate in L1, t_{L1miss} is the time to handle a miss in L1. Note that $t_{L1miss} = t_{L2hit} + \%_{L2miss} \times t_{L2miss}$ if there are two layers of cache. The formula is recursively defined depending on the number of layers of the cache. Here, with only one layer, we treat t_{L1miss} as a constant (1 in the theoretical analysis of I/O complexity) and we only investigate on the miss rate of L1, i.e. $\%_{L1miss}$, and the number of misses.

1.5 Cache parameters to simulate

We simulate and test algorithms in two settings:

1. Ideal cache model:
We use fully associative cache, with replacement policy LRU as an approximation of Belady's policy. Belady's policy is the theoretically optimal replacement policy which evicts a line that will be used furthest in the future. It is impractical to implement Belady's because it is a clairvoyant algorithm and it depends on the program being simulated.

2. Real-world caches:
It is usually atypical to implement fully-associative and LRU policy in the L1, because the circuit logic to handle fully-associative, i.e. compare all the tags in the cache in parallel, and to maintain the LRU information is too complex and costly, which will significantly increase t_{L1hit} which we hope to be small especially in L1. We will simulate the caches of the following four CPUs with NMRU replacement policy. The table below shows the parameters:

	Pentium III	MIPS10000	AMD Athlon	Itanium 2
L1 dcache size	16KB	32KB	128KB	32KB
L1 line size	32B	32B	64B	64B
L1 associativity	4-way	2-way	2-way	4-way
L1 number of sets	128	512	1024	128

2 Sorting

2.1 Merge Sort

2.1.1 Implementation

The merge sort algorithm is one of the most classical types of sorting algorithms. Our implementation of merge sort takes in a C-type integer array Arr, and two integer values: *start* and *end*, which serves as indexes to define the current interval of Arr. Note that the two indexes are inclusive, that is, they define an interval as $[begin, end]$, including the two endpoints.

After receiving the input values, the algorithm first finds the midpoint *mid* between *start* and *end*, and calls the merge sort algorithm between *start* and *mid*, and between $mid + 1$ and *end*. After the two intervals are sorted, the algorithm calls a two-way merge subroutine. The merge subroutine iterates through the two subarrays, each iteration picking the smallest element in the two arrays, and moving the element into the sorted array.

2.1.2 Experimental results

Merge Sort has the following theoretical I/O complexity bound:

$$Q(n) = 2Q(\frac{n}{2}) + Q^{\text{merge}}(n) = \Theta(\frac{n}{L}\log\frac{n}{Z})$$

where n is the array size, L is the line (block) size of the cache, and Z is the capacity of the cache.

For the ideal cache model (fully associative and LRU), the number of misses and miss rate as a function of array size is drawn below, with number of lines equal to 10, and line size (in byte) equal to 32, 64, 128.

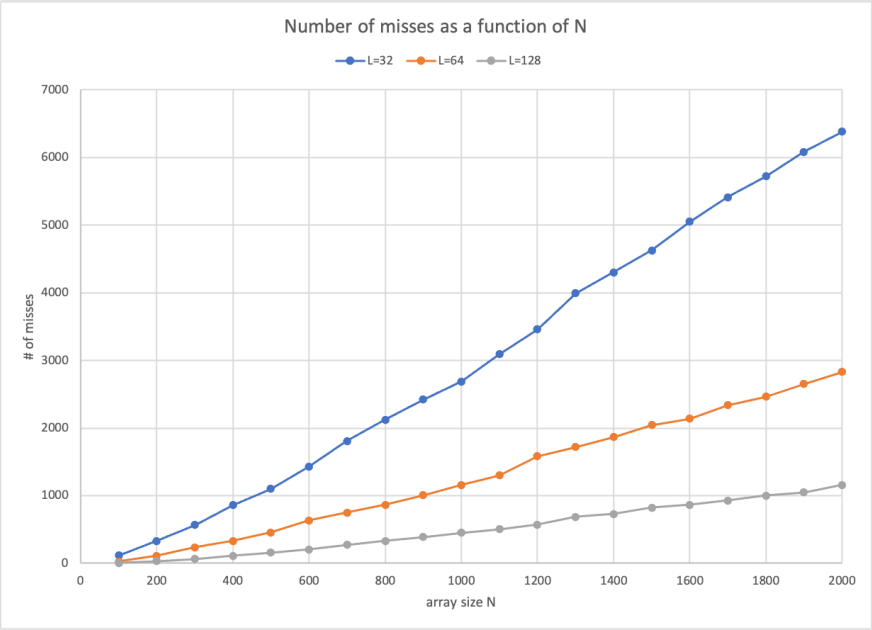

Figure 4: Merge Sort number of misses on ideal cache model

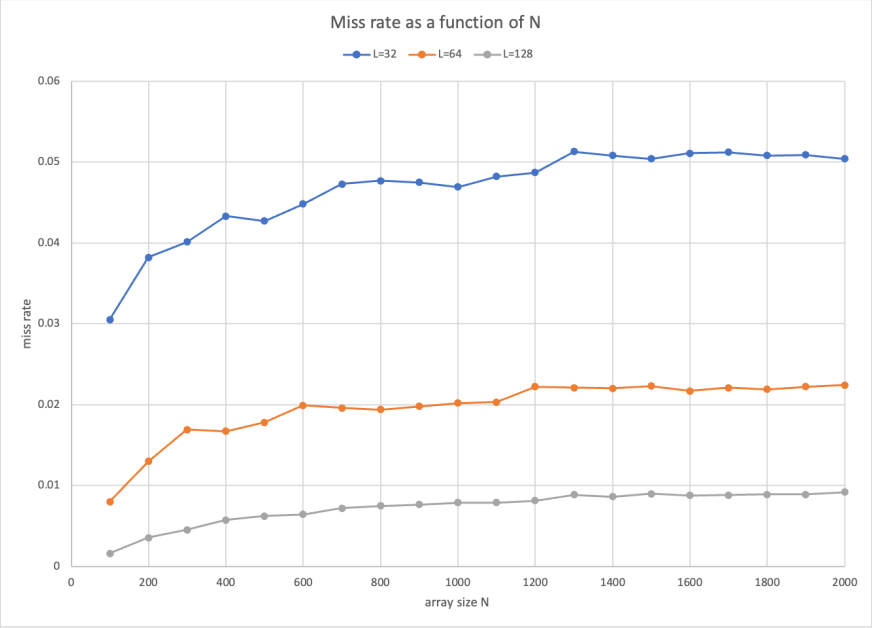

Figure 5: Merge Sort miss rate on ideal cache model

For the first graph, it can be observed that, with L and Z fixed, as the array size increases, the number of misses increases, resembling a linear function. As the parameter L increases, the number of misses decrease dramatically, roughly reversely proportional to L. Therefore, our testing results are basically consistent with the theoretical solution.

For the second graph, the miss rate slightly increases as n increases, and stabilizing at a certain value as array size increases. Similarly, the miss rate is also roughly reversely proportional to L.

For the real-world models (Pentium III, MIPS10000, AMD Athlon, Itanium 2), the number of misses and miss rate as a function of array size is drawn below:

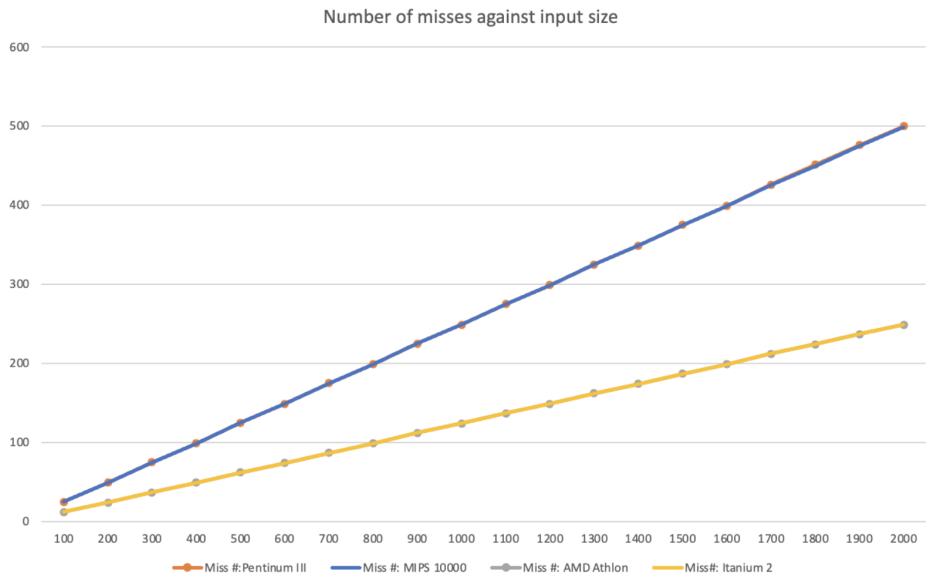

Figure 6: Merge Sort number of misses on real-world models

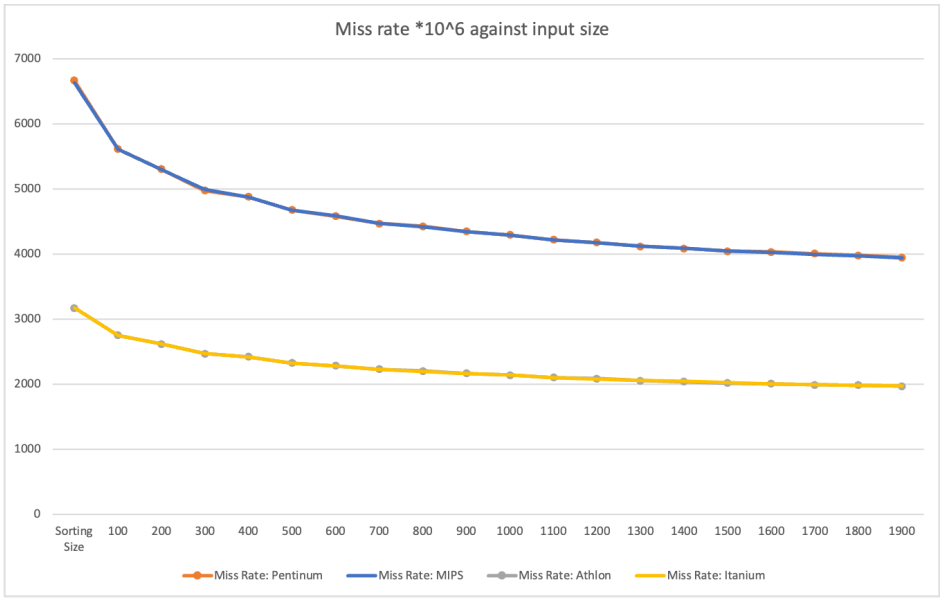

Figure 7: Merge Sort miss rate on real-world models

Note that in real-world cache options, the number of misses is roughly a linear line in all the models, increasing as the array size increases. The miss rates for all the cache options decrease as the array size increases, and then converges gradually. The number of misses and the miss rates for Pentium and MIPS, Athlon and Itanium, are very similar. This phenomenon is probably determined by the cache's line size, as Pentium and MIPS has $L = 32$, and Athlon and Itanium has $L = 64$. This reinforces our previous results regarding how L affects the miss rate and the number of misses.

2.2 K-way Merge Sort

2.2.1 Implementation

The K-way merge sorting algorithm is, in essence, similar to the merge sort algorithm. To perform the K-way merge, the integer array Arr is divided into K parts, and each part is sorted individually, by recursively calling the K-way merge sorting algorithm. After all the sub-arrays are sorted, the K-way merge is called, and the merge function merges the K sub-arrays, similar to the merge sort algorithm.

In our implementation, we use the naive K merging algorithm. The array Arr is divided into $\{A_1, A_2, \ldots, A_K\}$. Let B_i be the cache block associated with A_i, and let B be the cache block associated with Arr. Whenever a B_i is empty, it will be filled up with the next block from A_i. We keep transferring the next smallest element among all B_i's to B. Similarly, whenever B becomes full, it will be emptied by appending to Arr. In the ideal cache model, the block emptying and replacements will happen automatically. We use $K = \frac{Z}{L} - 1$ for the fully associative cache, where $\frac{Z}{L}$ is the number of lines, so that every block of the algorithm can fit into the cache. With this naive merging algorithm, $Q^{\text{merge}}(n) = O(K + \frac{n}{L})$

2.2.2 Experimental results

K-way Merge Sort has the following theoretical I/O complexity bound:

$$Q(n) = KQ(\frac{n}{K}) + Q^{\text{merge}}(n) = O(\frac{Kn}{Z} + \frac{n}{L}\log_K \frac{n}{Z})$$

where n is the array size, L is the line (block) size of the cache, and Z is the capacity of the cache.

For the ideal cache model (fully associative and LRU), the number of misses as a function of array size is drawn below, with number of lines E as 10, the K constant is set to 9, and line size (in byte) equal to 32, 64, 128.

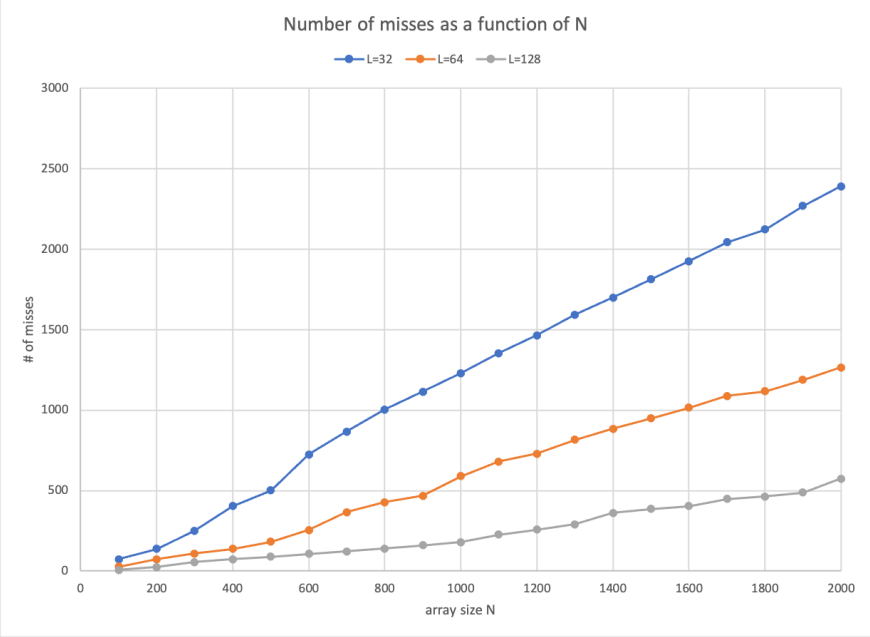

Figure 8: K-way Merge Sort number of misses on ideal cache model

From the graph, it can be observed that, with L, K and Z fixed, as the array size increases, the number of misses increases roughly in a linear function, with small deviations. Comparing across different lines, as the line size of the cache doubles, the number of misses roughly decreases by a half. Therefore, our testing results are basically consistent with the theoretical solution, and yields a similar performance pattern to the merge sort algorithm.

The following graph compares the K-way merge sort($K = 9$) with merge sort in the ideal cache model:

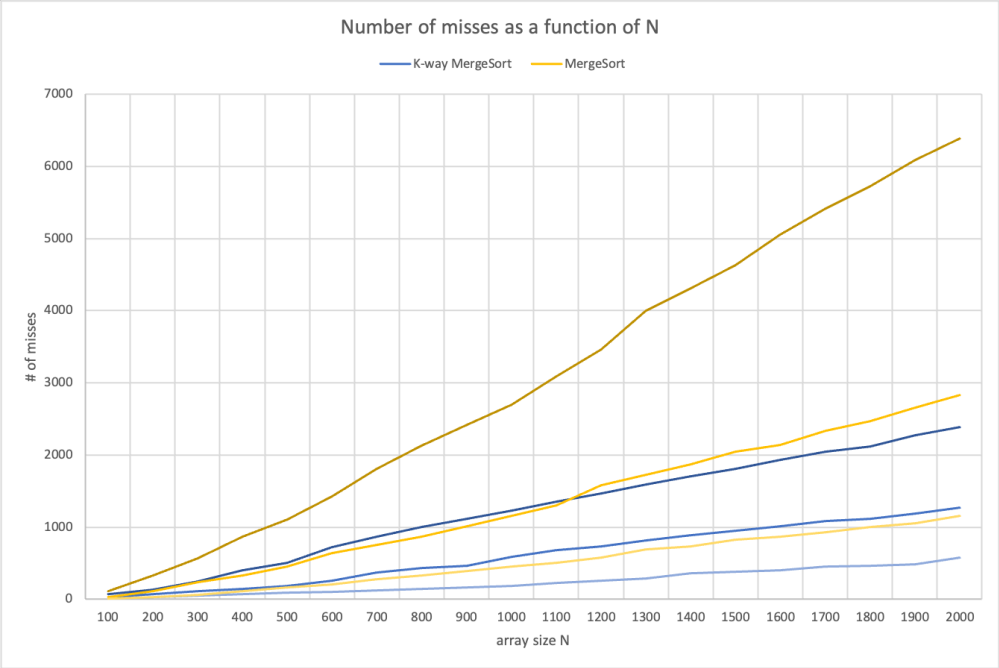

Figure 9: Merge Sort vs. K-way Merge Sort on ideal models

In the graph above, the yellow lines are merge sort and the blue lines are K-way merge sort. The same three cache parameters are simulated, represented by different darkness.

It can be concluded that in general, given the same L size, K-way merge sort performs more efficiently than merge sort. This is expected because with K-way merging, we have a log base K, rather than log base 2. From the graph, we can see that K-way merge sort is roughly two times faster (less number of misses) than merge sort, as we use $K = 9$. However, it can be seen that both algorithms have a similar asymptotic performance: when all the parameters are fixed, all the lines share the feature of $n \log n$.

For the real-world models (Pentium III, MIPS10000, AMD Athlon, Itanium 2), the number of misses and miss rate as a function of array size is drawn below: (in the real-world cache models, $K = 5$ was chosen)

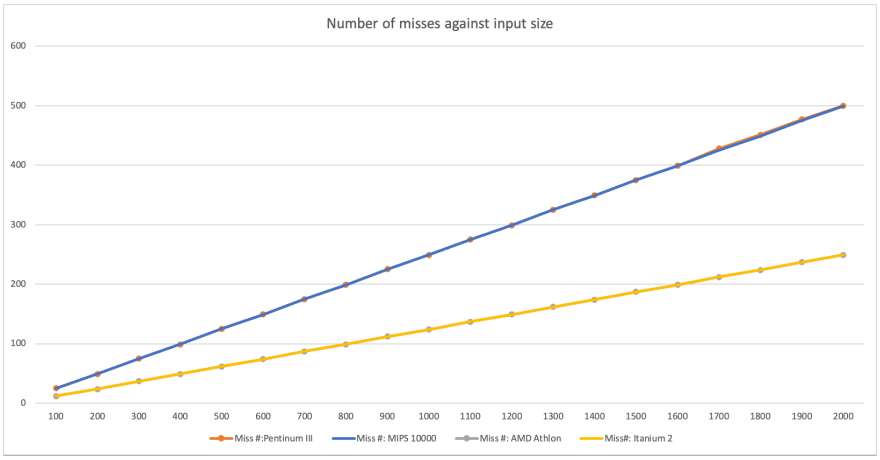

Figure 10: K-way Merge Sort number of misses on real-world models

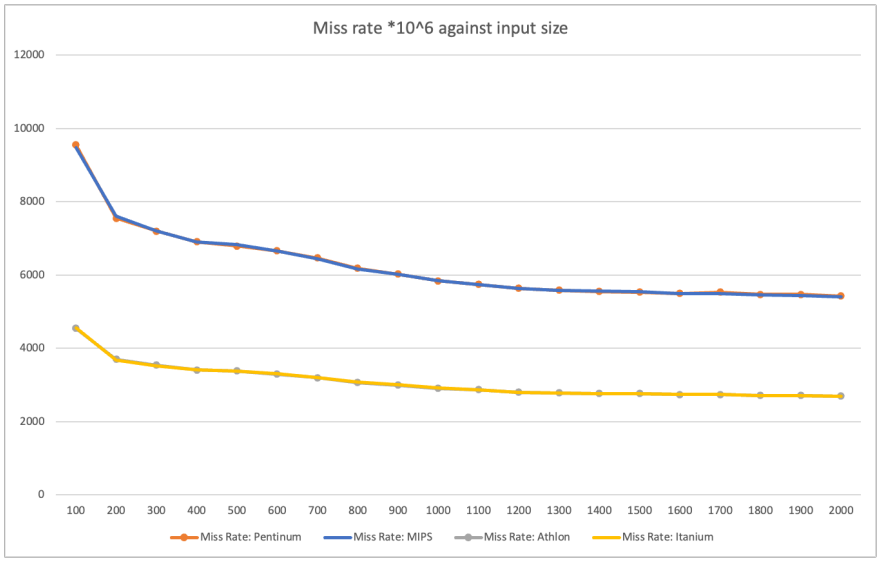

Figure 11: K-way Merge Sort miss rate on real-world models

It can be seen that similar to merge sort, the number of misses increases roughly in a linear line and the miss rate decreases, and then stabilizes at a certain level. Fluctuations are low, and the plot follows a clear trend line. The performance of Pentium/MIPS is still roughly two times worse than Athlon/Itanium, which is considered to be the result of different cache line sizes.

2.3 Funnel Sort

2.3.1 Implementation

The funnel sort algorithm implemented uses the following data structure definition:

```
struct Buffer
{ // FIFO queue
  int *data;
  int size; // max count of elements
  int head;
  int tail;
  int count; // current count of elements
};
```

Figure 12: struct Buffer

```
struct Funnel
{
  struct Buffer **in; // input arrays represented as buffers
  int in_count;       // count of input arrays (buffers) [n^1/3]

  struct Buffer **buffers; // intermediate buffers
  struct Funnel **left_fs; // left funnels
  int lf_count;           // count of left funnels = count of buffers

  struct Funnel *right_f; // right funnel
  struct Buffer *out; // output array
};
```

Figure 13: struct Funnel

The input arrays (of a funnel), the output array (of a funnel), and the intermediate buffers (inside a funnel) are all implemented as a FIFO queue, using the Buffer struct. The buffer struct has an integer array *data* that stores all the data, integer *size* and *count* to keep the size the the number of elements inside the FIFO queue, and also two pointers (index of the *data* array) *head* and *tail*. The buffer struct also has several functions associated with it:

enqueue: to put an element at *tail*.
dequeue: to get an element at *head*.
buffer_empty and buffer_full: return whether or not the buffer is empty (full).
buffers_empty: given an array of buffers, return whether or not all buffers are empty.
buffers_nonempty: given an array of buffers, return whether or not all buffers are nonempty.
get_buffer: given an array of buffers, return a non-empty buffer that has the smallest element at *head*.

The last three additional functions are useful for the implementation of the funnel sort.

For a funnel, it contains a list of input arrays (represented by an array of buffer pointers), a list of intermediate buffers, a list of left funnels, a right funnel, and an output array (represented by a buffer pointer).

For the funnels, conceptually there are two kinds of funnels:
red funnel: read inputs (dequeue) and finally write to the output (enqueue) even if there is an empty input buffer. It will read until all input buffers are empty (or have already output k^3 elements where k is the number of input arrays, here *in_count*).
green funnel: read inputs (dequeue) and finally write to the output (enqueue) if all buffers are nonempty. If one input buffer is empty after a read, it will call the funnel whose output buffer is this buffer. It will read until all input buffers are empty (or have already output k^3 elements where k is the number of input arrays, here *in_count*).

A base funnel is a funnel whose number of input buffers is less than some constant c. We use 5 in the experiments below. A base funnel will directly reads (dequeue) from input buffers and directly writes (enqueue) to the output buffer without the intermediate buffers, using the naive k-merging algorithm as the K-way merge sort.

When a funnel is called, it will first call all of its left funnels and then call its right funnel. If the funnel itself is red, then its left funnels are also red and its right funnel is green. If the funnel itself is green, then all of the smaller funnels it contains are green.

Figure 14: Recursive definition of the funnel structure

Here are some examples. If the whole funnel is just a one-layer funnel, in other words, the whole array can be divided into c input arrays where each array contains c^2 elements, then it must be a red funnel which does the naive k-merging. If the whole funnel is a funnel of two layers or three layers, it looks like the following:

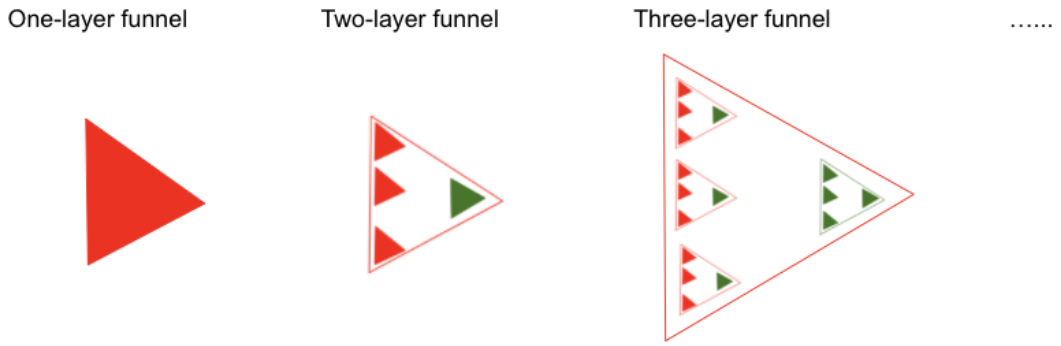

Figure 15: Funnel examples

2.3.2 Experimental results

Funnel Sort has the following theoretical I/O complexity bound:

$$Q(n) = n^{\frac{1}{3}} Q(n^{\frac{2}{3}}) + Q^{\text{merge}}(n^{\frac{1}{3}}) = O(\frac{n}{L} \log_Z n)$$

where n is the array size, L is the line (block) size of the cache, and Z is the capacity of the cache.

For the ideal cache model (fully associative and LRU), the number of misses and miss rate as a function of array size is drawn below, with number of lines and line size (in byte) equal to 128, 64; 64, 32; 64, 64.

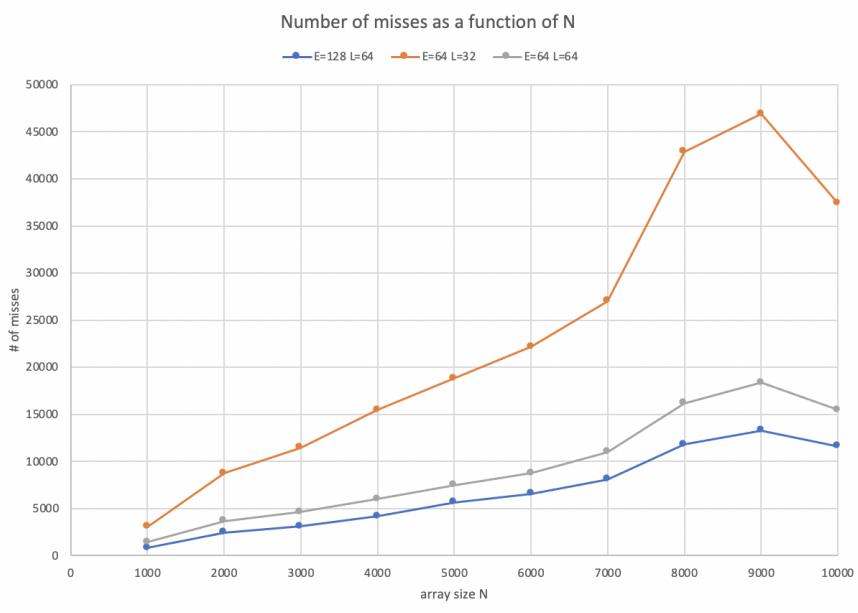

Figure 16: Funnel Sort number of misses on ideal cache model

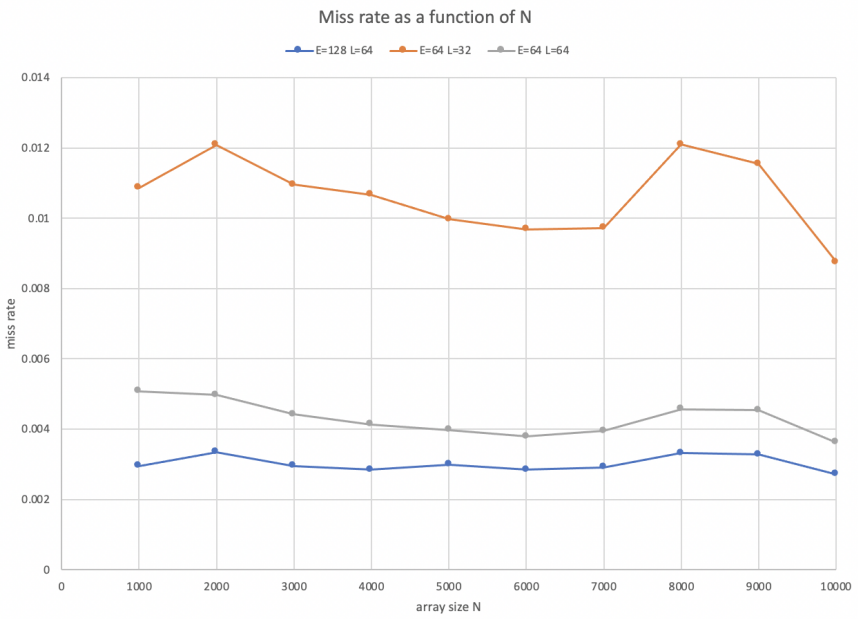

Figure 17: Funnel Sort miss rate on ideal cache model

For the first graph, it can be observed that, with L and Z fixed, as the array size increases, the number of misses increases slightly more than a linear fashion. Consider the orange line and the gray line, as the line size of the cache doubles, the number of misses roughly decreases by a half. Therefore, our testing results are basically consistent with the theoretical solution. For the second graph, the miss rate slightly decreases as n increases, but without a significant trend and is basically fluctuating.

For the real-world models (Pentium III, MIPS10000, AMD Athlon, Itanium 2), the number of misses and

miss rate as a function of array size is drawn below:

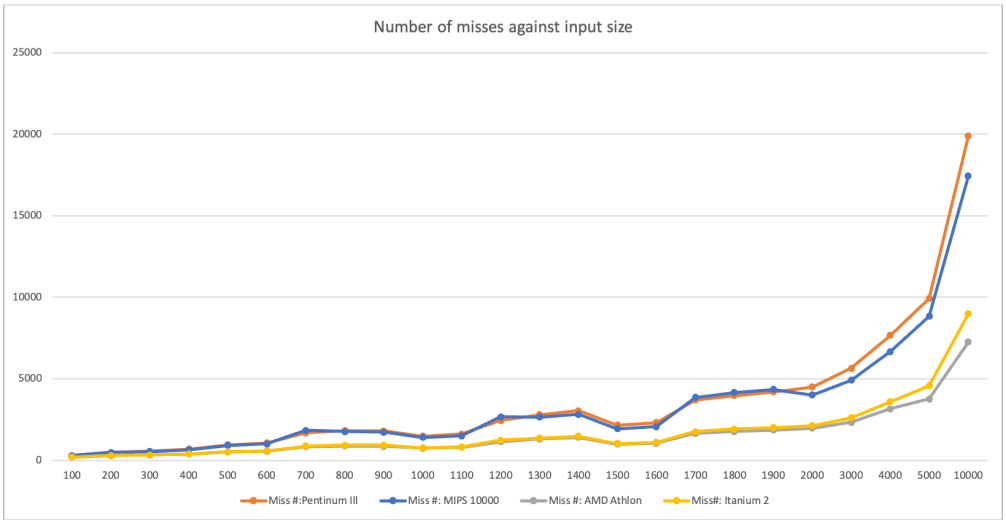

Figure 18: Funnel Sort number of misses on real-world models

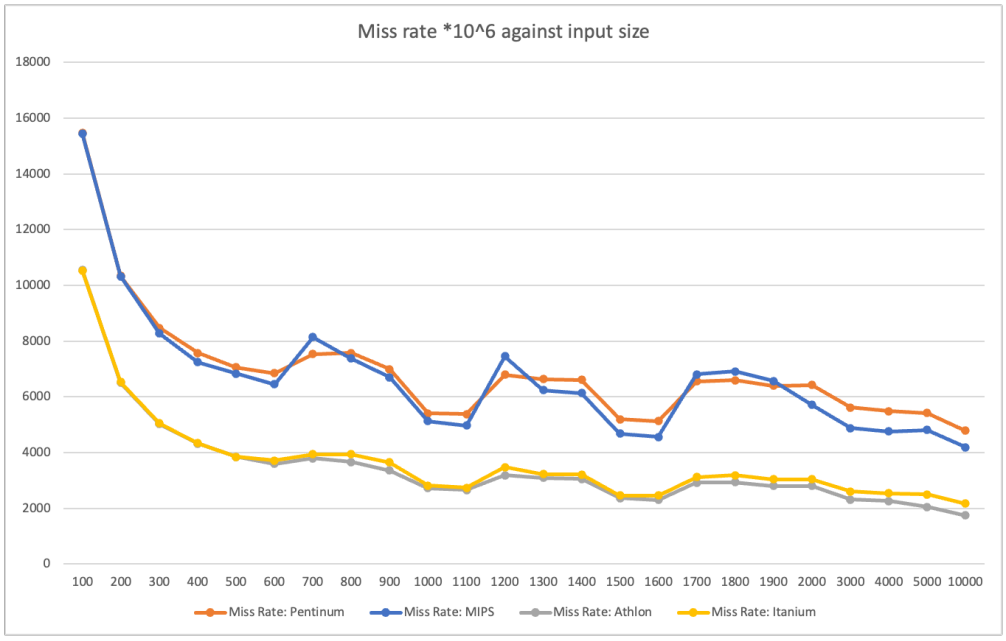

Figure 19: Funnel Sort miss rate on real-world models

It can be seen that there are some fluctuations around the increasing trend of the number of misses and the decreasing trend of the miss rate. The fluctuations are caused by the rounding issue of funnel sort. When the array size reaches some boundary, the funnel sort algorithm will create an additional funnel at a certain layer to handle the leftover.

Below is a comparison between K-way Merge Sort and Funnel Sort on the ideal cache model:

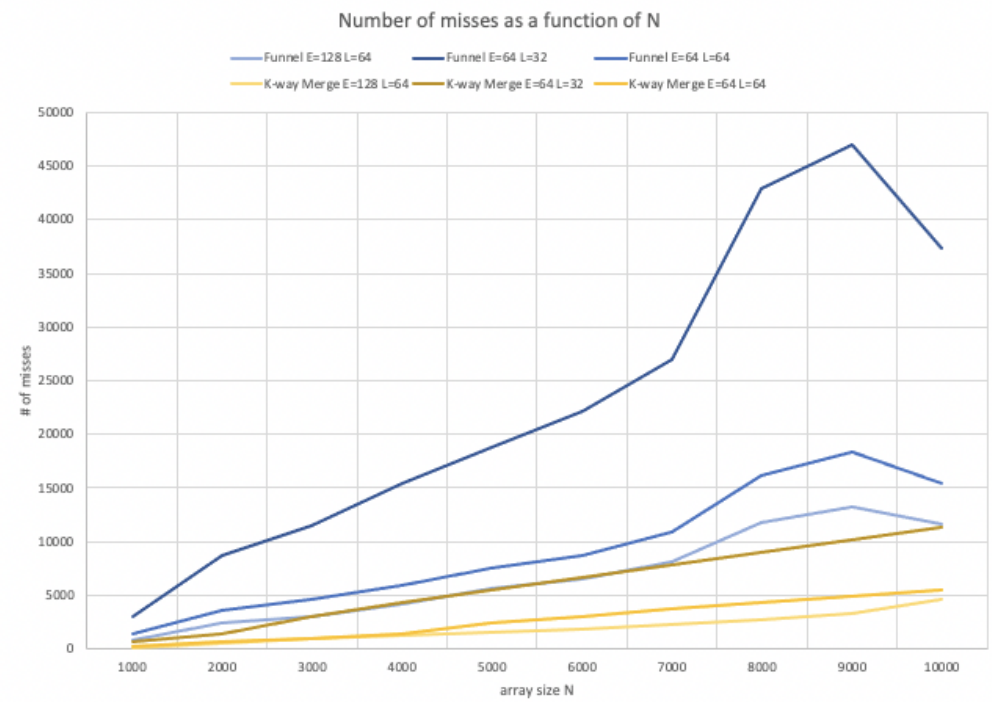

Figure 20: Funnel Sort vs. K-way Merge Sort on ideal models

In the graph above, the yellow lines are K-way merge sort and the blue lines are funnel sort. Three cache parameters are simulated, represented by different darkness. Our experimental results illustrate that for the same array size and the same cache parameters, K-way merge sort is roughly twice to three times as fast as the funnel sort. The additional cache misses of funnel sort are from many sources: first, K-way merge sort only needs to create a large temporary array once using malloc, whereas funnel sort needs to recursively create the funnels. Each time a funnel is created, malloc/free will be called which will add a constant number of cache misses each time. In addition, a one-layer funnel degrades to a naive k-merging algorithm, as illustrated in the implementation section, with a large overhead. As the number of layers increase, the overhead generated by the recursive data structure increases by a fair amount. Also, if the funnel has more than one layer, reading and writing in the intermediate buffer creates additional cache misses, compared to the K-way merge sort. Therefore, the simulation results we get for funnel sort and K-way merge sort are reasonable and as expected.

3 Matrix Multiplication

3.1 Blocked Multiplication

3.1.1 Implementation

Recall that blocked matrix multiplication is a cache-aware algorithm: it divides the matrix into multiple square blocks such that each block fits into the cache. Moreover, within each block, a naive n^3 matrix multiplication algorithm is used, in which 2 matrices are read and 1 matrix is written. The side length b of the block hence satisfies $3b^2 = Z$, which implies $b = \Theta(\sqrt{Z}) = c\sqrt{z}$, where $c = \frac{1}{\sqrt{3}}$.

The implementation is straightforward. There are 6 loops nested together: the outer 3 loops enumerate over all *blocks*, while the inner 3 loops perform the naive algorithm. For simplicity, we only test square matrices.

3.1.2 Experimental results

With tall cache assumption $Z = \Omega(L^2)$, cache-aware blocked multiplication achieves the optimal cache complexity:

$$Q(n) = \frac{Z}{L}\frac{n^3}{(c\sqrt{Z})^3} = \frac{n^3}{c^3\sqrt{Z}L}$$

We first run experiments by fixing n and Z while varying L. We set $n = 32$ integers long, $Z = 4096B$, and $L \in \{8, 16, 32, 64, 128\}$ bytes.

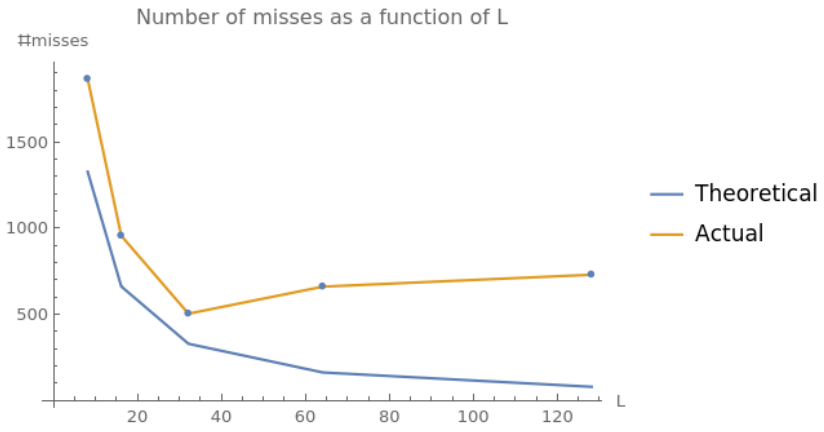

Figure 21: blocked multiplication #misses on ideal cache model with different L

When $L < 64$, the actual curve is closed to the theoretical one. When $L \geq 64$, we see some apparent difference between the two curves. This is because we fix Z and increase L, and when $L \geq 64$, tall cache assumption is broken. Our experimental results are still consistent with the theoretical ones.

Next we run experiments over real-world cache models. Since Z and L are fixed for each real-world cache model, we use different $n \in \{16, 32, 64, 96\}$.

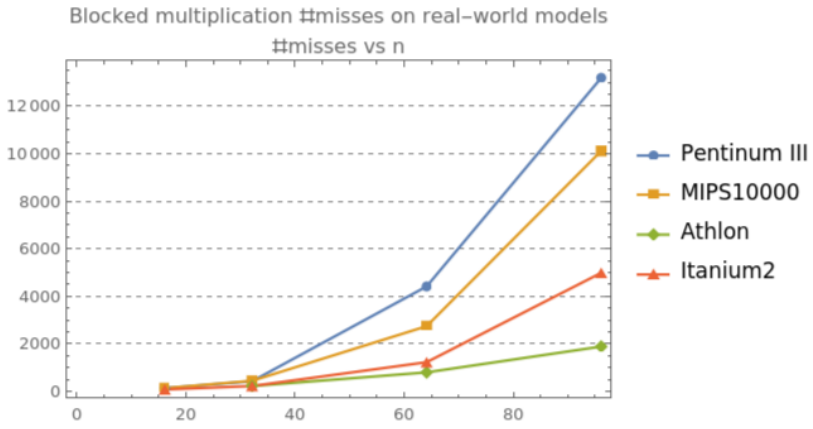

Figure 22: Blocked multiplication #misses on real-world models

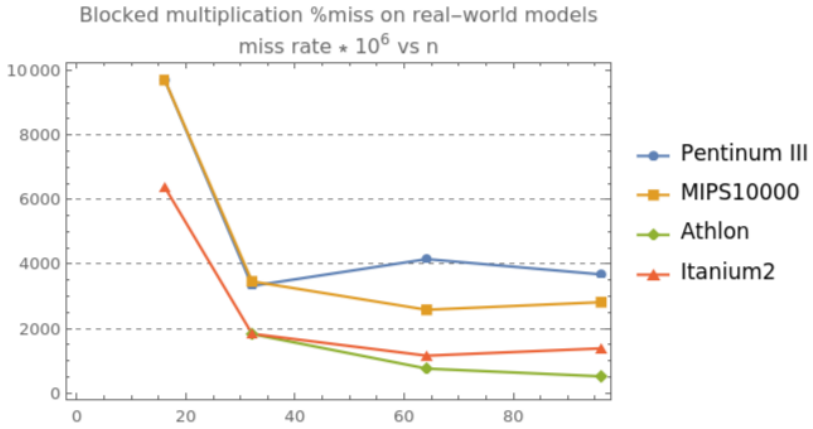

Figure 23: Blocked multiplication miss rate on real-world models

Similar to the sorting, as n increases, the miss rate decreases, with a little fluctuation. The performance of Pentium/MIPS is still about two times worse than Athlon/Itanium.

3.2 Morton-Z Layout Divide and Conquer

3.2.1 Implementation

The algorithm that accepts two input matrices A, B along with the address of the output matrix C, and calculates $C \mathrel{+}= A \cdot B$. (We have to first zero out in order to calculate $C = A \cdot B$.) We implement a divide-and-conquer matrix multiplication algorithm by dividing each matrix into 4 sub-matrices:

$$\begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix} \mathrel{+}= \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \cdot \begin{pmatrix} B_1 & B_3 \\ B_2 & B_4 \end{pmatrix}$$

$$\begin{aligned} C_1 &\mathrel{+}= A_1 B_1 + A_2 B_2 \\ C_2 &\mathrel{+}= A_1 B_3 + A_2 B_4 \\ C_3 &\mathrel{+}= A_3 B_1 + A_4 B_2 \\ C_4 &\mathrel{+}= A_3 B_3 + A_4 B_4 \end{aligned}$$

Each call requires 8 recursive matrix multiplications. Since addition is done in the base case, we don't need to do matrix addition explicitly. This results in a work recurrence of $T(n) = 8T(n/2) + O(1)$, which is $T(n) = \Theta(n^3)$. Therefore, there is no improvement in terms of work compared to the blocked matrix multiplication (which is also $\Theta(n^3)$). However, we pair this divide-and-conquer algorithm with Morton-Z layout, which will achieve the optimal cache complexity *without* the tall cache assumption while being *cache-oblivious*.

The algorithm assumes that both the input A, B and the output C are still row-majored, so it has to convert A, B into Morton-Z layout, do the recursive computation, and then convert the result back to row-major layout and store it in C. The skeleton of the algorithm is as follows:

```
void mult_dac(int N, const int A[N][N], const int B[N][N], int C[N][N])
{
    int mortonA[N*N], mortonB[N*N], mortonC[N*N];
    memset(mortonC, 0, sizeof(int)*N*N); // zero-out output matrix
    orig2morton(N, mortonA, A); // convert A to Morton Z-layout
    orig2morton(N, mortonB, B); // convert B to Morton
    dac(N, mortonA, mortonB, mortonC); // divide and conquer
    morton2orig(N, mortonC, C); // convert the result to row-major layout
}
```

The mapping between row-major layout and Morton-Z layout is done by bits interleaving, as illustrated below:

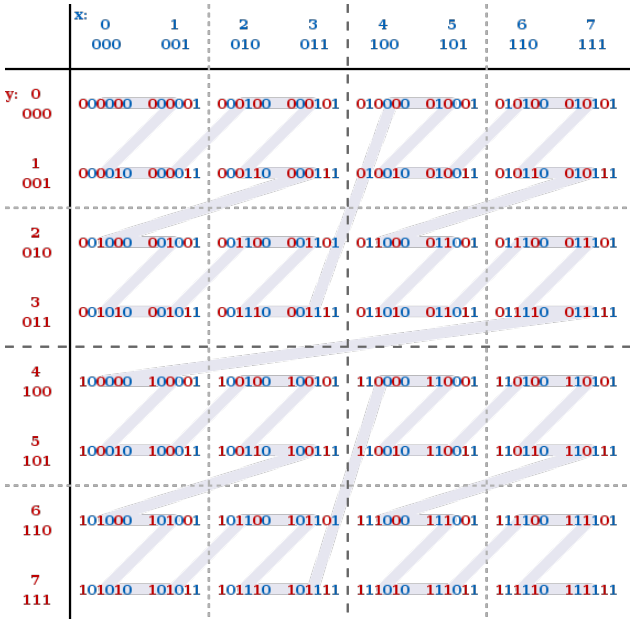

Figure 24: Each entry at (x, y) corresponds its index in the Morton Z-layout

3.2.2 Experimental results

We verify the cache complexity $Q(n) = \dfrac{n^3}{c^3\sqrt{Z}L}$ by fixing n and Z while varying L. Specifically, we set $n = 32$ integers long, $Z = 4096B$, and $L \in \{8, 16, 32, 64, 128\}$ bytes.

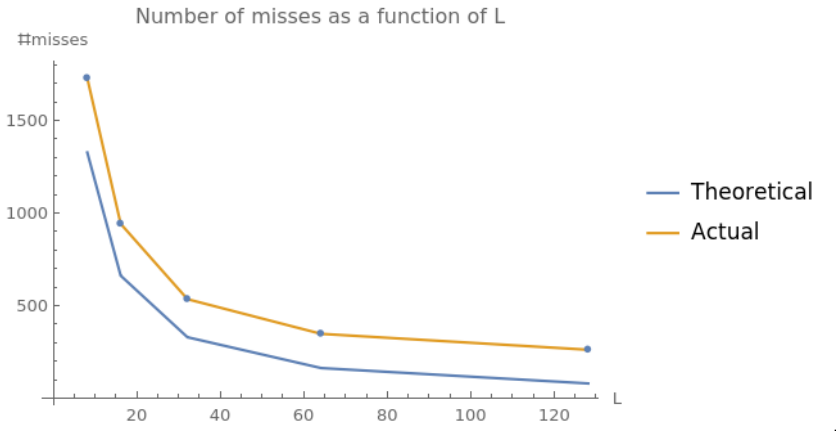

Figure 25: Z-layout multiplication number of misses on different L

The actual curve perfectly follows the trend of the theoretical curve. Because the code performs layout conversion before and after the main divide-and-conquer algorithm, which incurs cache misses that are not captured in the theoretical bound, the actual number of misses is always higher than the theoretical value.

Note that when $L = 128$B, where the tall cache assumption is broken, the actual curve is still following the theoretical one. This is expected as divide-and-conquer in Morton-Z layout does not require the tall cache assumption.

Next we run experiments over real-world cache models. One downside of our implementation of Morton-Z layout is that it only supports a side-length that is a power of 2, so we choose $n \in \{16, 32, 64, 128\}$.

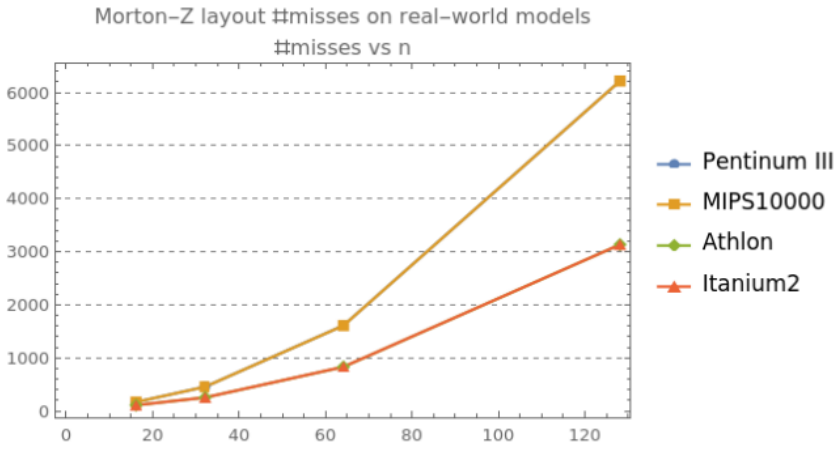

Figure 26: Z-layout multiplication number of misses on real-world cache models

(Pentinum III overlaps with MIPS 10000. Athlon overlaps with Itanium 2)

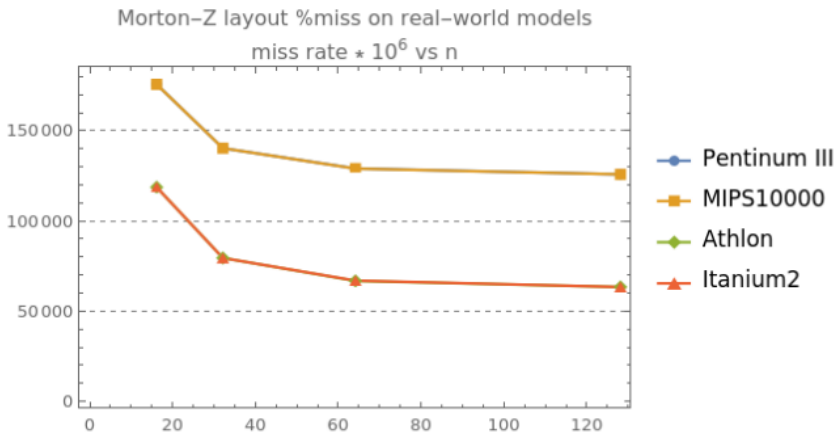

Figure 27: Z-layout multiplication miss rate on real-world cache models

Similar to blocked multiplication and sorting, as n increases, the miss rate decreases, with a little fluctuation. The performance of Pentium/MIPS is still about two times worse than Athlon/Itanium.

4 Conclusion

We've developed a cache simulator that is capable of handling

- various evictions policies.
- various algorithms written in C, with correctness checking and cache performance logging.
- various one-level cache configurations.

In both sorting and matrix multiplication scenarios, we implemented several algorithms whose actual number of misses is largely consistent with the theoretical model; the only case in which they differ is when tall cache assumption is broken in blocked matrix multiplication experiments. In real-world cache models, we see similar trends for all algorithms and processors: while n increases, miss rate decreases. Pentinum III seems very similar to MIPS 10000, while Athlon seems similar to Itanium 2, in terms of their cache performance on the algorithms we implemented, and this is likely because Athlon and Itanium 2 have a larger block size of 2^6 bytes while Pentinum III and MIPS 10000 have a smaller block size of 2^5 bytes.

5 References

- cache simulator: CSAPP cachelab (CSE361S)
- Morton-Z Layout: CSE539S course slides
- various theoretical bounds: CSE549T lecture notes